**UVVM Essential Mechanisms** –Quick Reference  
  
This document explains some of the essential mechanisms necessary for running UVVM, in addition to helpful and important VVC status and configuration records which are accessible directly from the testbench. A more comprehensive review can be found in the VVC Framework Manual.

[1 Libraries 1](#_Toc17902587)

[2 UVVM Initialization 2](#_Toc17902588)

[3 UVVM and VVC Shared Variables 3](#_Toc17902589)

[4 VVC Status, Configuration and Transaction information 4](#_Toc17902590)

[5 Direct Transaction Transfer 4](#_Toc17902591)

[5.1 Transaction definitions 4](#_Toc17902592)

[5.2 Transaction details 4](#_Toc17902593)

[6 Multiple Central Sequencers 4](#_Toc17902594)

[7 Compile scripts 5](#_Toc17902595)

[8 Scope of verbosity control 5](#_Toc17902596)



# Libraries

In order to use a VVC the following libraries need to be included:

library uvvm\_util;

context uvvm\_util.uvvm\_util\_context;

library uvvm\_vvc\_framework;

use uvvm\_vvc\_framework.ti\_vvc\_framework\_support\_pkg.all;

library bitvis\_vip\_<name>;

context bitvis\_vip\_<name>.vvc\_context;

# UVVM Initialization

The following mechanisms are required for running UVVM

|  |  |
| --- | --- |
| **Mechanism** | **Description** |
| **ti\_uvvm\_engine** | **ti\_uvvm\_engine**  This entity contains a process that will initialize the UVVM environment, and has to be instantiated in the testbench harness, or alternatively in the top-level testbench.  Example:  i\_ti\_uvvm\_engine : entity uvvm\_vvc\_framework.ti\_uvvm\_engine; |
| **await\_uvvm\_initialization()** | **await\_uvvm\_initialization(VOID)**  This procedure is a blocking procedure that has to be called from the testbench sequencer, prior to any VVC calls, to ensure that the UVVM engine has been initialized and is ready. This procedure will check the shared\_uvvm\_state on each delta cycle until the UVVM engine has been initialized.  Note that this method is depending on the ti\_uvvm\_engine mechanism.  Note that this method uses the t\_void parameter, defined in the UVVM Utility Library types package.  Example:  await\_uvvm\_initialization(VOID); |

# UVVM and VVC Shared Variables

UVVM and VVC shared variables are defined in global\_signals\_and\_shared\_variables\_pkg and the various vvc\_methods\_pkg, respectively.

Shared variables

|  |  |
| --- | --- |
| **Shared variable** | **Description** |
| shared\_uvvm\_status | Shared variable providing access to VVC related information via the info\_on\_finishing\_await\_any\_completion record element, i.e.  shared\_uvvm\_status.info\_on\_finishing\_await\_any\_completion  This record element gives access to the name, command index and the time of completion of the VVC that first fulfilled the await\_any\_completion(). The available record fields are:  vvc\_name : string -- default “no await\_any\_completion() yet”  vvc\_cmd\_idx : natural -- default 0  vvc\_time\_of\_completion : time -- default 0 ns  For more information regarding other fields available in the shared\_uvvm\_status see the UVVM Util QuickRef, section 1.4 |
| shared\_<vvc\_name>\_vvc\_config | Shared variable providing access to configuration parameters for each VVC instance and channel if applicable. E.g.  shared\_sbi\_vvc\_config(1).inter\_bfm\_delay.delay\_type := TIME\_START2START;  shared\_uart\_vvc\_config(RX,1).bfm\_config.bit\_time := C\_BIT\_TIME; |
| shared\_<vvc\_name>\_vvc\_status | Shared variable providing access to status parameters for each VVC instance and channel if applicable. E.g.  v\_num\_pending\_cmds := shared\_sbi\_vvc\_status(1).pending\_cmd\_cnt;  v\_current\_cmd\_idx := shared\_uart\_vvc\_status(TX,2).current\_cmd\_idx;  v\_previous\_cmd\_idx := shared\_uart\_vvc\_status(TX,2).previous\_cmd\_idx; |
| shared\_<vvc\_name>\_transaction\_info | Shared variable providing access to VVC instances transaction information to include in the wave view during simulation.  Available information is dependent on VVC type and typical information is:  operation : t\_operation; -- default NO\_OPERATION  data : std\_logic\_vector(C\_VVC\_CMD\_DATA\_MAX\_LENGTH-1 downto 0); -- default 0x0  msg : string(1 to C\_VVC\_CMD\_STRING\_MAX\_LENGTH); -- default empty |

# VVC Status, Configuration and Transaction information

The VVC status, configuration and transaction information records are defined in each individual VVC methods package.

Each VVC instance and channel can be configured and useful information can be accessed from the testbench via dedicated shared variables.

From the VVC configuration shared variable, one is given the ability to tailor each VVC to one’s needs, in addition to access the BFM configuration record via the bfm\_config identifier. In addition to BFM configuration possibility, the configuration settings consist of command and result queue settings, BFM access separation delay and a VVC dedicated message ID panel.  
Note that some BFMs require user configuration, e.g. the bit\_time setting in serial interface BFMs.

The VVC status shared variable provide access to the command status parameters for each of the VVCs, such as the current and previous command index, and the number of pending commands in the VVCs command queue. This provide a helpful tool, e.g. when synchronizing VVCs in the test sequencer using the await\_completion() or await\_any\_completion() methods.

When using a wave viewer during simulation, the transaction shared variable provides helpful information regarding current VVC operation and transaction information such as address and data. Note that the accessible fields depend on the VVC and its implementation. An example of two SBI VVCs performing FIFO write operations, followed by check operations, is shown in Figure 4‑1.



Figure 4‑1 VVC Transaction info example

# Direct Transaction Transfer – From BFMs, VVCs and/or Monitors

UVVM now supports sharing transaction information in a controlled manner within the complete testbench environment. This allows BFMs, VVCs and Monitors to provide transaction info to any other part of your testbench – using a predefined structured mechanism. This makes it even easier to make good VHDL testbenches.

Transaction information may be used in many different ways, but the main purpose is to share information inside the testbench of activity or accesses on a given DUT interface. Normally such information is provided from a dedicated interface monitor, but making such a dedicated monitor is sometimes quite time consuming and often not really needed. For that reason UVVM provides a mechanism for getting the transaction information directly from the BFM or VVC. The BFM in this case includes monitor capabilities, but this is a very small overhead as the BFM handles the interface protocol in detail anyway. As the VVC uses the BFM for handling the interface, the VVC may appear as the source of the transaction information.

## Purpose

The purpose of the direct transaction transfer is to allow a model or other part of the testbench to see exactly what accesses have been made on the various interfaces of the DUT, so that the expected DUT behaviour and outputs may be checked. Let us illustrate this with a really simple testbench scenario to verify a UART peripheral with an AXI-lite register/CPU interface on one side and the UART RX and TX ports on the other side. The test sequencer may command the AXI-lite BFM or VVC to write a data byte into the UART TX register, and then it must be checked that the data byte is transmitted out on the DUT TX output some time later.

1. A simple testbench approach would be to have the test sequencer also telling the receiving UART BFM or VVC exactly what to expect. This is a straight forward approach, but does require more action and data control inside the test sequencer. That could of course all be handled in a super-procedure, but for any undetermined behaviour inside the BFM or VVC, like random data generation or error injection, this would not work.
2. A more advanced approach is to have a model overlooking the DUT accesses, generate the expected data and tell the receiving BFM or VVC to check for that data
3. An even more advanced approach would be to use a Scoreboard to check received data (from DUT via VVC) against expected data from a model.

However, for the two latter approaches the model needs information about exactly what happened (the transaction) on the various DUT interfaces, so that it can generated the correct expected data. For the model it doesn’t matter if the transaction info comes from a monitor or from a BFM/VVC, as long as the information is correct.

The model could of course look at the interfaces and analyse the transactions itself, but distributing this task to the BFM, VVC or monitor makes the testbench far more structured and significantly improves overview, maintenance, extensibility and reuse – at least for anything above medium simple verification challenges.

Another purpose of providing transaction information is for progress viewing and debugging – typically via the wave view or simulation transcripts.

## Transaction definitions

By transactions we normally talk about a complete end to end transfer of data across an interface. This could be anything from a simple write, read or transfer of a single word - to a complete packet in a packet-oriented protocol like Ethernet. The word transaction is however also used for both sub-sets and super-sets of this – depending on the protocol and even on how we want to control our system or testbench. In order to communicate properly and to assure that transactions are properly understood, the following terms are defined:

* **Base transaction (BT)** is the lowest level of a complete transaction as allowed from the central sequencer.   
  E.g. AXI-lite read, write or check, UART transmit, receive or expect, Ethernet transmit, receive or expect
* **Sub-transaction (ST)** is the lowest level of an incomplete transaction as allowed from a BFM. The sub-transaction as such is complete seen from a handshake point of view, but the transfer of data is not complete. Typical for a split transaction protocol.  
  E.g. Avalon Read Request and Avalon Read Response
* **Leaf transaction (LT)** is the lowest level of complete **or** incomplete transaction defined for a given protocol. I.e. a sub-transaction when this is defined for a given protocol, otherwise a base transaction.  
  (e.g. for Avalon: the sub transactions, and for UART, SBI or Ethernet: the base transactions (as no sub-transaction exist for these protocols)
* **Compound transaction (CT)** is a set of transactions or other methods or statements that as a total is doing a more complex operation.   
  E.g. SBI\_Poll\_Until() or a UART transmit of N consecutive bytes.

## Transaction information

Information about the above transactions is typically provided to a model in the test harness. Depending on whether the transaction info is provided from the BFM, VVC or Monitor, different types of information will be available. Common for all is that they always provide info about the operation (read, write, transmit, etc) and often also any other protocol specific info. For a UART this could be data and parity, for an SBI it could be address and data, and for Ethernet: the packet fields.

This minimum is normally what the Monitor can provide from just analysing the interface, and this is also normally enough for a model to generate expected DUT behaviour and outputs. The BFM and VVC on the other hand, can provide more info, which could be useful for instance for progress viewing and debugging.

The transaction information is organised as a transaction record with some predefined fields as shown below. The first table shows the general transaction record, whereas the second table shows a concrete example for the SBI.

Note that for a given interface/protocol, the BFM, VVC and Monitor will use the same interface dedicated transaction record type – with some fields potentially unused.

Table 1 – Transaction record t\_transaction (General). The greyed-out fields indicate optional or protocol dedicated fields

|  |  |  |
| --- | --- | --- |
| **Field name** | **Type name** | **Description** |
| Operation | t\_operation | Protocol operation on the given DUT interface. E.g. NO\_OPERATION, WRITE, READ, TRANSMIT, POLL\_UNTIL, …  NO\_OPERATION is default and thus used when there is no access. All operations will be separated with a NO\_OPERATION for at least 1 delta cycle, e.g. NO\_OPERATION – WRITE – NO\_OPERATION – READ – NO\_OPERATION. |
| <Optional protocol dedicated field(s)> | <Protocol dedicated> | One or more fields required to complete the transaction info E.g. for UART: Single field ‘data’; for SBI: field 1: ‘addr’, field 2: ‘data’; for Ethernet: Most ethernet fields as separate fields here – or a better solution include as a complete sub-record |
| Transaction\_status | t\_transaction\_status | Transaction status (‘FAILED’ or ‘SUCCEEDED’), but will show ‘IN\_PROGRESS’ until status is known, and for a complete CT where success cannot be known. Between transactions: ‘NONE’ |
| Meta | t\_vvc\_meta | Additional transaction information – only known by the VVC and/or BFM.  ‘msg’ is known by both, whereas ‘cmd\_idx’ is only known by the VVC (will be set to -1) A monitor has no knowledge of this as and will set them to msg = “”, cmd\_idx = -1 |
| Error\_info | t\_error\_info | Any type of error injection relevant for the given protocol. Typically parity or stop-bit error in an UART or a CRC error in an Ethernet. If no error injection or detection has been implemented, this sub-record may be left empty. |
| NOTES:  - For BFM and VVC transaction info the record reflects the current status on the interface (as the BFM and VVC know this from transaction start, whereas the Monitor will set up the record only after knowing it has failed or succeeded. | | |

Table 2 – Transaction record t\_transaction (For SBI). The greyed-out fields indicate protocol dedicated fields

|  |  |  |
| --- | --- | --- |
| **Field name** | **Type name** | **Description** |
| Operation | t\_operation | Either of WRITE, READ or CHECK, but could also be POLL\_UNTIL or a more complex compound transaction |
| Address | unsigned |  |
| Data | std\_logic\_vector |  |
| Transaction\_status | t\_transaction\_status |  |
| Meta | t\_vvc\_meta |  |
| Note: No error\_info field as no error injection or detection has been implemented in neither BFM, VVC nor Monitor. | | |

Other interfaces will of course have different protocol dedicated fields, or even a complete protocol dedicated sub-record (e.g. for Ethernet packet fields)

## Transaction info transfer

In order to reduce the number of signals from a VVC or Monitor, all possible simultaneous transactions (and their transaction records) are collected into a single transaction group record.

For an SBI interface this will consist of a BT record and potentially a CT record, whereas for an Avalon it will in addition also consist of two ST records because for instance a read request may be active at the same time as a read response (and the sub-transactions are part of a base transaction and may also be part of a CT).

Table 3 below shows the maximum transaction group record for an SBI, whereas Table 4 below shows the maximum transaction group record for an Avalon. The greyed-out CT is optional for both, and thus depends on whether CTs have been defined (in the VVC). Multiple parallel STs may be written to the transaction group record simultaneously – as these are handled by different “threads” (concurrent statements).

A Monitor cannot know about CTs, and thus a monitor will never fill inn that sub-record. A Monitor for a split transaction protocol (i.e. with multiple STs) may or may provide BT info. If it does, this should normally be implemented in a higher level “wrapper”

A BFM procedure can only provide information about a single leaf transaction, as sub-transactions should normally be implemented in different processes (and thus different BFM procedures) in order to operate simultaneously. A BFM procedure will update the relevant transaction record inside the transaction group record.

Note again:  
- A VVC will update its DTT details during the transaction. It will be set as the BFM is called, changed when status is known, and turned off when BFM is finished.

- A monitor will set its DTT information after the transaction is finished (and keep it on for a pre-defined time – or when the next transaction is finished)

It is recommended that the model (or any other user of the DTT signals) triggers on transaction\_status changing to ‘NONE’

Table 3 – Maximum transaction group record t\_transaction\_group – for an SBI interface

|  |  |  |
| --- | --- | --- |
| **Field name** | **Type name** | **Description** |
| bt | t\_transaction | Base transaction |
| ct | t\_transaction | Compound transaction |

Table 4 - Maximum transaction group record t\_transaction\_group – for an Avalon MM interface

|  |  |  |
| --- | --- | --- |
| **Field name** | **Type name** | **Description** |
| st\_request | t\_transaction | Sub-transaction |
| st\_response | t\_transaction | Sub-transaction |
| Bt | t\_transaction | Base transaction |
| ct | t\_transaction | Compound transaction |

## Transaction info transfer signals

The DTT (Direct Transaction Transfer) is provided out of the BFM/VVC/Monitor using global signals. These signals and all DTT related VHDL types are defined in the transaction\_pkg..

- Monitor DTT signal : global\_<protocol-name>\_monitor\_transaction, e.g. global\_uart\_monitor\_transaction

- VVC DTT signal: global\_<protocol-name>\_transaction, e.g. global\_uart\_transaction. The VVC is also responsible for filling out the vvc\_meta record field. (always updated inside the actual BFM inside the VVC)

# Multiple Central Sequencers

A structured test environment is important and we recommend the use of a test harness to instantiate VVCs, DUT, clock generator and so forth. The testbench may consist of one or more test sequencers which are used to control the complete testbench architecture with any number of VVCs, although for a better testbench overview we recommend to have a single central test sequencer only.

# Compile scripts

In the script folder in the root directory the compile\_all.do compiles all UVVM components. This script may be called with one to three input arguments. The first input argument is the directory of the script folder at the root directory from the working directory. The second input argument is the target directory of the compiled libraries, by default every library is compiled in a sim folder in the corresponding components directory. The third input argument is the directory to a custom component list in .txt format. The script will only compile the components listed in that file. By default, the script uses the file component\_list.txt located in uvvm/script. This file can be modified so that only some components are compiled.

Example: do uvvm/script/compile\_all.do uvvm/script

There are also compile scripts for all UVVM components located in the script folder of each UVVM component. These scripts can be called with two input arguments. The first input argument is the directory of the component folder from the working directory. The second input argument is the target directory of the compiled library, default is the sim folder in the respective component.

Example: do uvvm/uvvm\_util/script/compile\_src.do uvvm/uvvm\_util

# Scope of verbosity control

Message IDs are used for verbosity control in many of the procedures and functions in UVVM, as well as log messages and checks in VVCs, BFMs and Scoreboards.   
Note that VVCs and Scoreboards comes with dedicated message ID panels and are not affected by the global message ID panel, but accessed by addressing targeting VVC or Scoreboard and, if applicable, instance number or with a broadcast.

E.g.

-- Global message ID panel. Does not apply to VVCs or Scoreboards, as they have their own local message ID panel

disable\_log\_msg(ALL\_MESSAGES);

enable\_log\_msg(ID\_SEQUENCER);

-- VVC message ID panel  
disable\_log\_msg(VVC\_BROADCAST, ALL\_MESSAGES); -- broadcast to all VVCs and instances

enable\_ log\_msg(I2C\_VVCT, C\_VVC\_INSTANCE\_1, ID\_BFM\_WAIT); -- I2C VVC instance 1  
enable\_log\_msg(I2C\_VVTC, C\_VVC\_INSTANCE\_2, ID\_BFM\_WAIT); -- I2C VVC instance 2

-- Scoreboard message ID panel

shared variable sb\_under\_test : record\_sb\_pkg.t\_generic\_sb;

…

sb\_under\_test.disable\_log\_msg(ALL\_INSTANCES, ID\_CTRL); -- broadcast to all SB instances

sb\_under\_test.enable\_log\_msg(C\_SB\_INSTANCE\_1, ID\_DATA); -- SB instance 1

A subset of message IDs is listed in UVVM Utility Library QR, section 1.10.

Disclaimer: This IP and any part thereof are provided "as is", without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement.  
In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with this IP.

**INTELLECTUAL**

**PROPERTY**